UNIVERSITY OF CALIFORNIA
SANTA CRUZ

CLOUD INTERFACE FOR LIVEHD

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Rohan Jobanputra

December 2019

The Thesis of Rohan Jobanputra
is approved:

Professor Jose Renau, Chair

Professor Scott Beamer

Professor Chen Qian

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Cloud Interface for LiveHD

by

Rohan Jobanputra

There is an increasing demand in the world of cloud computing and for good reason. Cloud computing has opened up a world of possibilities including easy web hosting, fast computation, and the ability to store unlimited data on the cloud. Gone are the days where a user would have to buy expensive hardware to meet their computing needs.

LiveHD is an open-source framework designed for fast simulation and synthesis. It supports multiple hardware description languages such as Verilog and Pyrope. It also supports other open-source tools such as Yosys, ABC, OpenTimer, and Mockturtle.

My small contribution to the project was to figure out a way to use the power of cloud computing to produce these simulation and synthesis results. There is a promising open-source tool called *gg*, which is being developed by Stanford University. This tool makes it easy to build and run applications on the cloud, so it would make sense to use gg with LiveHD. This process of integration will be discussed in this thesis.

Acknowledgments

Firstly, I'd like to thank my advisor Professor Jose Renau for allowing me to work in his lab. I took his logic design class with Verilog (CMPE 125) when I was an undergraduate student at UC Santa Cruz, and I learned a lot from that class. That was one of the primary reasons why I asked him to be my graduate advisor. I want to thank him for all his hard work and his continuous support throughout my graduate career.

I'd also like to thank Professor Scott Beamer and Professor Chen Qian at UC Santa Cruz for providing feedback for this thesis and helping me improve on it.

I'd like to thank all my friends at UC Santa Cruz who helped me throughout my undergraduate and graduate career: Pranav, Tanvir, Wuyuan, Rho and Harsh. I'm grateful to have them in my life, and I couldn't have done this without them.

Special thanks for Sadjad Foulad for answering my questions about *gg*.

I'd like to thank my parents for their continuous support, and all the sacrifices they have had to make to ensure their children could live a better life. Lastly, I'd like to thank my brother, Anand, without whom, none of this would have been possible. We grew up together, and he has always been there for me through thick and thin and has helped me achieve many of my personal goals. He is not only the best brother anyone could ask for, but also a great mentor, coach, and a really good friend. I hope I can repay him someday.

Chapter 1

Introduction

Do. Or do not. There is no try.

Yoda

The current set of hardware synthesis tools take a lot of time to synthesize designs even for minor changes in the design flow. Sometimes it might take hours for even the smallest of changes. This is not very efficient for engineers as they would have to wait a long time to see the results and fix their bugs accordingly.

LiveHD aims to improve on that metric by producing synthesis results in a couple of seconds. LiveHD is an open-source graph library developed for live hardware development based on an incremental synthesis model. LiveHD consists of multiple components, the main ones being LGraph (Live Graph) and LNAST (Language neutral AST) [1].

Traditionally, if a programmer wanted to take advantage of more computing power for their applications, they would have to either physically walk into a data center (in the pre-internet era), or log into a server and use their resources. This was not the most efficient way for users to scale up their applications because it would take up too much time and/or cost a lot of money. The internet has opened

up a countless number of possibilities, one of them is cloud computing. In today's world, the need for cloud computing has increased drastically, mainly because it offers better scalability, better security, and is easier to use than traditional data centers. According to a 2019 study, 83 percent of enterprise workloads will be on the cloud by 2020, and 94 percent of enterprises already use a cloud service today [2]. This goes to show the increasing rate of demand and adoption for cloud computing.

Since the goal of LiveHD is to produce synthesis and simulation results within a few seconds, it makes sense to use cloud computing in order to speed up the process. However, this process is not easy and poses a few challenges. The main challenge is to account for all dependencies LiveHD needs to make sure it works on the cloud. These dependencies include all the libraries needed for LiveHD and its modules as well as all the input and output files. Accounting for these is challenging and it will be discussed in Chapter 4.

Chapter 2 will cover LiveHD and its internals, Chapter 3 will discuss different cloud technologies offered today, and finally, the best way to implement LiveHD functionalities on the cloud will be discussed in Chapter 4.

Chapter 2

LiveHD

Always pass on what you have learned.

Yoda

This chapter is about LiveHD which is an open-source graph library that allows the user to represent different phases of the synthesis and physical design flow [3]. LiveHD aims to optimize the live hardware design process by producing synthesis and simulation results within a few seconds.

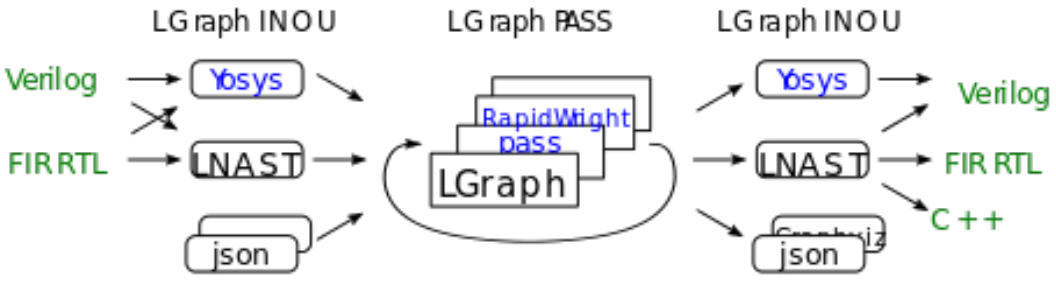

Figure 2.1: LiveHD Framework [1].

Figure 2.1 is a high-level block diagram of the LiveHD framework. Its core component is LGraph, which is a bi-directional graph structure, which means that

it allows forward and backward traversals in the nodes. The graph is based on numbered nodes, edges, and a tabular structure. Nodes may contain many inputs and outputs, just like any other graph.

2.1 Introduction

The goal of the entire LiveHD project was to create an open-source tool that could represent different stages of the digital design flow. LiveHD can represent netlists, RTL designs as well as place and route designs. It also takes in Verilog, Liberty, LEF/DEF, and also Pyrope (a modern hardware description language) files as inputs.

No such open-source project exists at the moment, however, there are different tools to represent the design flow separately. For example, Verilog and BLIF (Berkeley Logic Exchange Format) are used during logic synthesis, LEF/DEF (Liberty exchange format / Design exchange format) is used during physical design, and GDS is used for layout. [3]

LiveHD uses a custom shell interface, lgshell, which allows the user to interact with LiveHD, as well as provide inputs to and generate outputs from LiveHD.

2.2 Design Flow

Figure 2.2 shows the flow when an LGraph is generated, starting with an RTL Description like Verilog or Pyrope, and then goes through different passes (transformations).

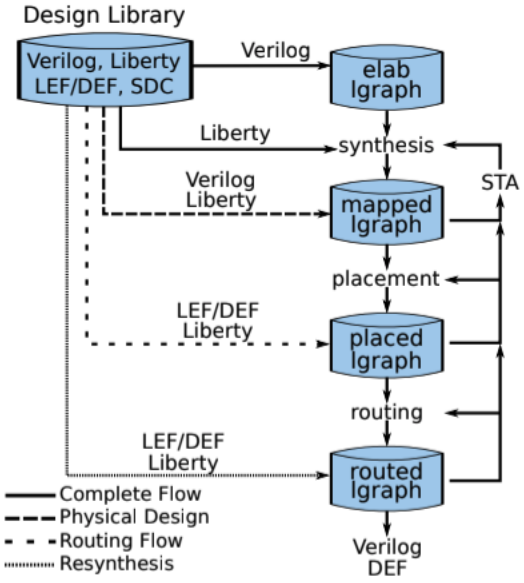

Figure 2.2: The overall RTL flow when an LGraph is generated [3].

Users have the option going through the entire flow (synthesis, placement, routing, timing) or they can also start down the flow using any netlist [3].

2.3 LGraph Database

The LGraph database is primarily built on a memory map structure. This allows for fast transformations. "Conceptually, the database contains a target technology (for mapped designs) and a set of modules (LGraphs) that represent the design itself. Information corresponding to the standard cell library is not duplicated in the graph nodes; instead, each node has a type that points to a specific cell in the library." [3]

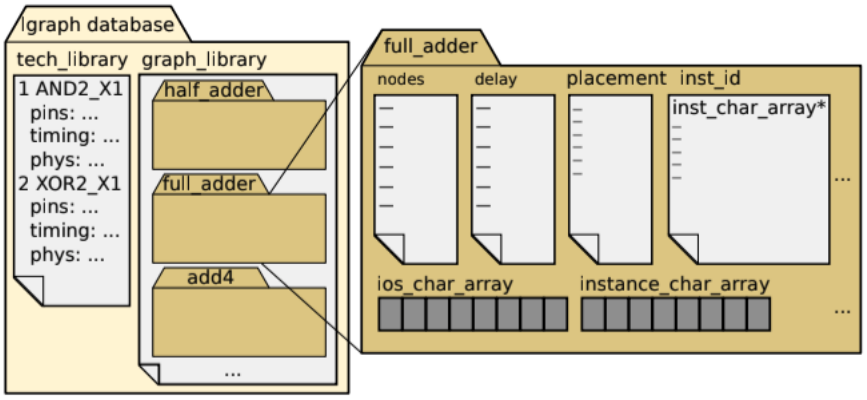

Figure 2.3: The LGraph Database [3].

Figure 2.3 shows that the database is a collection of LGraphs, and each LGraph is made up of indexed tables. Each index is represented by a node ID. Users can also define their custom tables if there is a need for LGraph to interact with new applications.

2.4 Graph Representation

LGraph is a bi-directional graph which means that it allows for backward and forward traversals. As mentioned in the previous section, the graph consists of nodes, node pins, and edges. This is what enables the traversal in the graph. This is represented with an adjacency list as shown in Figure 2.4, and the advantage of using that is the graph is more memory efficient.

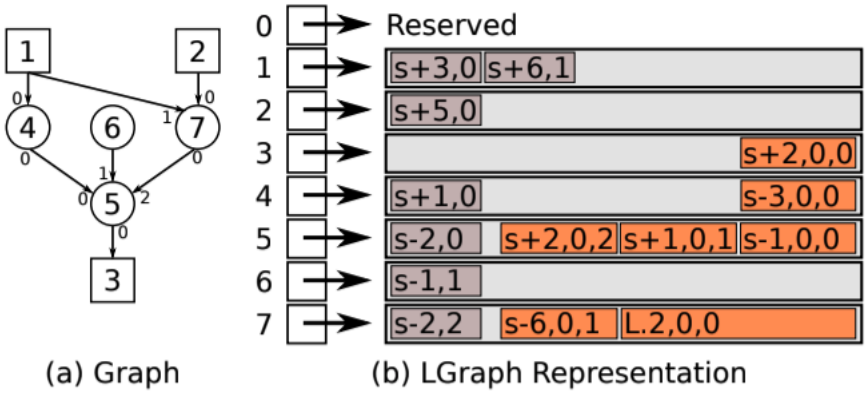

Figure 2.4: The LGraph adjacency list [3].

In Figure 2.4 the graph on the left is translated to an LGraph which is shown on the right. Netlists are represented by multiple graph nodes, and each gate in the design is given a unique node identification number (Node ID), and a port in the gate is given a port ID.

2.5 Interface

LiveHD uses a custom interactive shell, lgshell, which allows the user to interact with the project. Some of the more important commands are shown in table 2.1.

Commands	Function
inou.yosys.tolg	read verilog using yosys to lgraph
inou.yosys.fromlg	write verilog using yosys from lgraph
lgraph.open	open an lgraph if it exists
lgraph.stats	print the stats from the passed graphs
inou.cfg.tolg	generate an lgraph from a cfg (pyrope)
inou.liveparse	liveparse and chunkify verilog/pyrope files
lgraph.liberty	add liberty files to the lgraph library
lgraph.dump	verbose insides for lgraph
lgraph.stats	print the stats from the passed graphs

Table 2.1: Some LGraph commands and their functionality.

The "InOu" command is short for input/output and lets the user pass inputs to LGraph or generate outputs from it. For example, the command "inou.yosys.tolg" converts a Verilog file to an LGraph representation.

```
1 module trivial( input a, input b, output c);
2 assign c = a ^ b;
3 endmodule
```

Figure 2.5: A simple Verilog Example.

The example shown in Figure 2.5 sets the output (c) to the 1-bit xor of the two inputs (a and b).

```
lgraph> inou.yosys.tolg files:./inou/yosys/tests/trivial.v
```

Figure 2.6: An InOu yosys command which transforms a Verilog file to an LGraph database.

The command in Figure 2.6 takes the trivial.v file and converts it to an LGraph database. The database files are generated in the current folder under the "lgdb" directory.

2.6 Conclusion

In this chapter, the internals of LiveHD and its structure was discussed, along with the custom shell used to interact with it called lgshell.

This was just one of the examples of an lgshell command, but several other passes can be done as indicated from the figure above. The next chapter will discuss different cloud technologies currently available and also go over a potential method to perform graph transformations on the cloud.

Chapter 3

Background

> Train yourself to let go of
> everything you fear to lose.
>
> Yoda

This chapter will discuss the different build tools needed for LiveHD as well as different cloud services offered today. I tried different approaches for this project, and they will be discussed in Chapter 3 and Chapter 4.

3.1 What Is Cloud Computing?

Before discussing the different methods to run applications on the cloud, what exactly is cloud computing? Microsoft answered this question perfectly. "Simply put, cloud computing is the delivery of computing services including servers, storage, databases, networking, software, analytics, and intelligence over the Internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change." [4]

One of the primary benefits of using cloud services is that users can take advantage of the thousands of servers available, thus speeding up their applications immensely. Users just pay for their computation needs, which means they do not have to invest in expensive hardware, and they just pay for how many times they use a cloud compute engine or pay according to what their storage needs are. Most cloud services have a free option as well, which is good for users who are just getting started and learning about cloud computing.

Since most cloud servers may not necessarily have the libraries needed to run user applications, they must be containerized (packaged) in such a way so that all the dependencies needed to run these applications on the cloud are included. The following two sections will discuss the most popular methods to package applications today.

3.2 Docker

Docker is currently one of the most popular cross-platform tools which is used to package applications and deploy them on the cloud. Docker is compatible with most cloud services such as Google Cloud, Amazon Web Services (AWS), and Microsoft Azure.

3.2.1 What Is Docker?

"Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux

machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code." [5]

As stated above, Docker containers consist of the application itself (sometimes multiple applications), plus all of the libraries needed to run the application, which is why it can be run on any system which supports Docker, hence Docker containers are easier to deploy on the cloud.

3.2.2 How Does It Work?

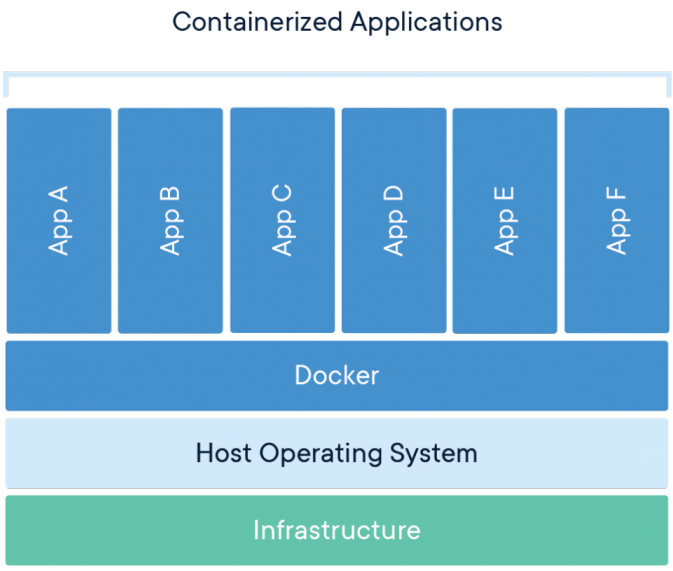

Figure 3.1: An example of a Docker Container [6].

Figure 3.1 is a high-level diagram of what a Docker container looks like. On the bottom, there is a host kernel, and top of that, the host operating system (Windows, Mac OS, Linux). The application level is where the user installs the Docker application on their host computer, and this allows the containerization

of applications. Containers separate the application from its host so that it can be run on virtually any system that can run Docker. Docker almost works like a virtual machine, with the key difference being that Docker does not need a host operating system to run the application.

Once a Docker image has been created, the user has the option to publish their image on Docker Hub. By doing so, any user anywhere in the world will have access to this image, and all they have to do is download the Docker image to run the application.

3.2.3 Advantages Of Using Docker

- Works on most operating systems
- Better reproducibility as applications work exactly the same on all systems.
- Docker Hub is a free repository to upload Docker images, and any public image can be downloaded by any user.
- Fast deployment because it's a self-contained system.

3.2.4 Disadvantages Of Using Docker

- Applications cannot be run at bare metal speeds because of containerization.
- Not easily scalable mainly due to the reason above.
- Data in the container can be lost if the container goes down.

3.3 Kubernetes

Kubernetes is an open-source containerization orchestrator made by Google and works well with the Google Cloud Engine. This is different than a containerization platform like Docker because Kubernetes can essentially manage multiple, complicated containers.

In a bigger production environment, there would be a lot more containers, and Kubernetes makes sure that these containers are running as they should, without much human effort. It also makes it easier to scale applications, either up or down.

3.3.1 How Does It Work?

Kubernetes is based on the traditional client/server model. It consists of a Master node and worker nodes. These nodes are further subdivided into different components. Figure 3.2 is a high-level diagram of the Kubernetes Architecture which shows what nodes are made of how the master and worker nodes interact.

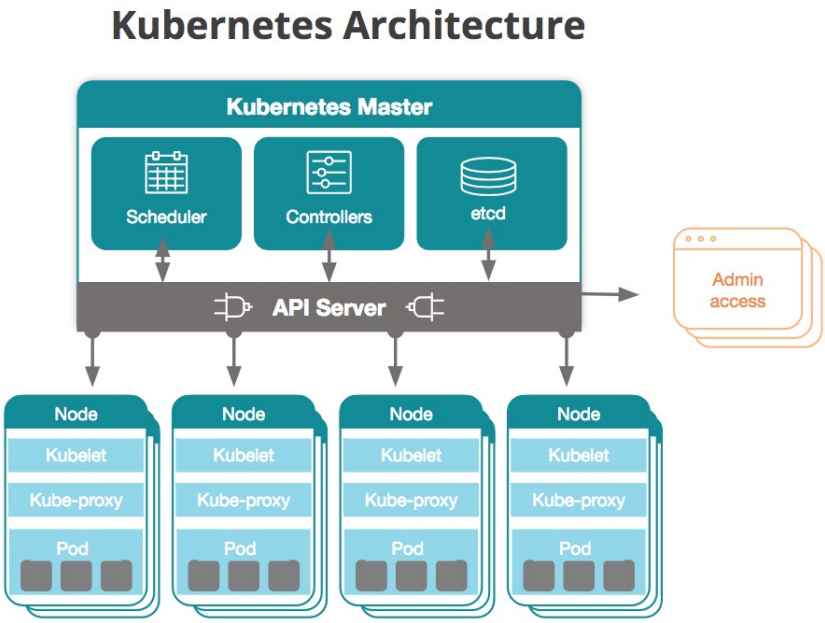

Figure 3.2: Kubernetes Architecture [7].

- **Nodes**: A node is a worker machine for Kubernetes. This could either be a VM or physical machine and caters to all the tasks that need to be done to run applications. The master node controls all the other nodes [8].
- **Master**: The master node controls all the other nodes, and perform all the assigned tasks [9].
- **Pods**: Pods are containers that can be deployed to the node. These contain the necessary data and also the IP addresses needed to run applications [9].
- **Replication controller**: This would control and manage duplicate pods[9].
- **Service**: This handles service requests made by nodes to other nodes and pods[9].

- **Kubelet**: This is a service that runs on all the nodes and makes sure that containers are up and running[9].

- **Kubectl**: This is the Kubernetes command-line tool which allows the user to configure everything while using Kubernetes [9].

A cluster is made up of all the nodes needed to run an application. As seen in Figure 3.2, each node consists of containers (pods) and its corresponding service (kubelet). These children nodes can be thought of as the data plane. The master node communicates with the other nodes and according to the user input, sends service requests. Users only interact with the master node and this can be thought of as the control plane of the application.

3.3.2 Advantages of Using Kubernetes

- Highly Scalable.

- Better version control. This makes it easier to update applications on the fly due to microservices.

- Self-healing, which means it can detect failing containers and automatically deploy backups.

- Seamless integration with Google Cloud.

3.3.3 Disadvantages Of Using Kubernetes

- Big learning curve for new users. This might reduce productivity.

- It could be more expensive to use for users who only want to run a single application on the cloud.

3.3.4 Difference Between Kubernetes And Docker

Table 3.1 shows the key differences between Kubernetes and Docker.

Docker	Kubernetes
Docker is a container platform for building, configuring and distributing Docker containers.	Kubernetes is an ecosystem for managing a cluster of Docker containers known as Pods.
Docker uses its very own native clustering solution for Docker containers called Docker Swarm.	Kubernetes is not a complete solution and uses custom plugins to extend its functionality.
The load balancer is deployed on its own single-node swarm when pods in the container are defined as a service.	Load balancing comes out of the box in Kubernetes because of its architecture and it's very convenient.
Setup is quick and easy compared to Kubernetes.	Takes relatively more time for installation.

Table 3.1: Kubernetes vs Docker [10].

On a more basic level, Kubernetes can be thought of as a ship, and Docker can be thought of as the cargo it's carrying. Docker can package the applications into containers (cargo), and Kubernetes manages them (the ship).

3.4 Google Cloud vs AWS

Currently, Google Cloud and Amazon Web Services (AWS) are two popular cloud services providers. They offer similar services, such as similar storage and compute engines, but there are a few key differences as shown in Table 3.2.

<table>
<tr><th>AWS</th><th>Google Cloud</th></tr>
<tr><td>Support for Windows and Linux systems, GPU support, and automatic scaling.</td><td colspan="2">Support for Windows and Linux.</td></tr>
<tr><td>Support for instances with up to 128 virtual CPU's and 3904GB of RAM.</td><td>Support for instances with up to 96 virtual CPU's and 624GB RAM.</td></tr>
<tr><td>Storage volume sizes from 1GB to 16TB</td><td>Storage volume sizes from 1GB to 64TB</td></tr>
<tr><td>Available in 21 regions.</td><td>Available in 20 regions.</td></tr>
<tr><td>The max object size for a single upload is 5GB.</td><td rowspan="2">Objects can be up to 5TB for both single and multi-part upload.</td></tr>
<tr><td>The max object size for a multi-part upload is 5TB.</td></tr>
<tr><td colspan="2">Both have a pay per second pricing model and offer free trials.</td></tr>
</table>

Table 3.2: AWS vs Google Cloud [11].

AWS has a slight edge over Google Cloud, but the main advantage of using AWS is its autoscaling feature and its compute options. AWS's free tier is perfect for beginners because it allows for more compute time every month. AWS also has a bigger market share than its competitors (33 percent). This is one of the reasons why it has better support and adoption rates.

3.5 Wake

Wake is an open-source build orchestration tool and custom language developed by SiFive [12]. It is optimized for fast builds due to build-caching and is a good alternative to make and Bazel.

3.5.1 How Does It work?

- By initializing Wake in the current working directory, it generates a database (Wake.db) which keeps a record of the state of the build.
- Whenever Wake is run, it checks that database first to see the status of the build.
- First, it builds a list of dependent jobs. Some jobs may depend on other jobs, and this dependency list helps in figuring out the order of execution.
- After the list is built, Wake performs an in-depth dependency analysis. If all the input files are under-specified, the build always fails. If they are over-specified, Wake does not use the unused dependencies.
- Wake automatically detects parallelism and serializes jobs that are dependent on each other.
- Wake then copies the pre-built files if they exist, or builds new ones if needed.

3.5.2 Advantages Of Using Wake

- Wake keeps a record of all the build actions so errors can be easily traceable.
- Support for parallelism.

- Shared build caching. There could be multiple contributors to a project, and Wake allows the use of pre-built files for faster builds.

3.6 Bazel

Bazel is another open-source build system that supports C++, Java, Android, iOS, Go, and many other language platforms[13]. Bazel needs a BUILD file that contains the sources, dependencies, and targets, just like any other build system.

3.6.1 How Does It Work?

- The first requirement is to have a BUILD file written in Starlark, which is a domain-specific language [13].
- All the inputs, targets, and build rules are specified in this file.
- Bazel begins the build process by loading the BUILD file and analyzing the target [13].
- It then checks all the dependencies and applies all the build rules, and produces an action graph [13].
- The action graph is a model which consists of build artifacts, how they are related, and the steps needed to build a target [13].
- The final step is to execute the build and generate the output [13].

3.6.2 Advantages Of Using Bazel

- Speeds up builds with advanced caching.
- Support for multiple languages.

- Works with most popular operating systems such as Linux, Mac OS, and Windows.
- Highly scalable and it handles codebases of all sizes.

Just like Wake, it supports caching, which improves build performance, which is why LiveHD is built using Bazel.

3.7 *gg* - The Stanford Build System

This section is about *gg*, which is a new build system developed by Stanford University. This framework and a set of command-line tools help users build and run common applications, and does this in such a way that the application can be built on systems other than the host machine, like the cloud. This allows the application to be built faster since the user can take advantage of more processing power available on other machines since *gg* allows for parallel processing.

While *gg* is still in active development, it aims to achieve building applications on the cloud with support for up to 10,000 parallel cloud functions. By using *gg*, the user can reduce costs because the need for constantly running clusters on a server would be eliminated. *gg* also allows pre-built applications to be executed using its engine. This can be done locally or on the cloud.

3.7.1 How Does It Work?

On a high-level, *gg* creates a model of all the dependencies needed to build and run a program, and generates an executable called a "thunk". "In the functional-programming literature, a thunk is a parameterless closure (a function) that captures a snapshot of its arguments and environment for later evaluation. The

process of evaluating the thunk-applying the function to its arguments and saving the result-is called forcing it" [14].

gg uses a method called model substitution, in which it creates a model (thunk) containing the dependencies, and then uses that model to run the application. The *gg* engine can be used to run the thunk (using the command "gg force"), and this process will generate the required output.

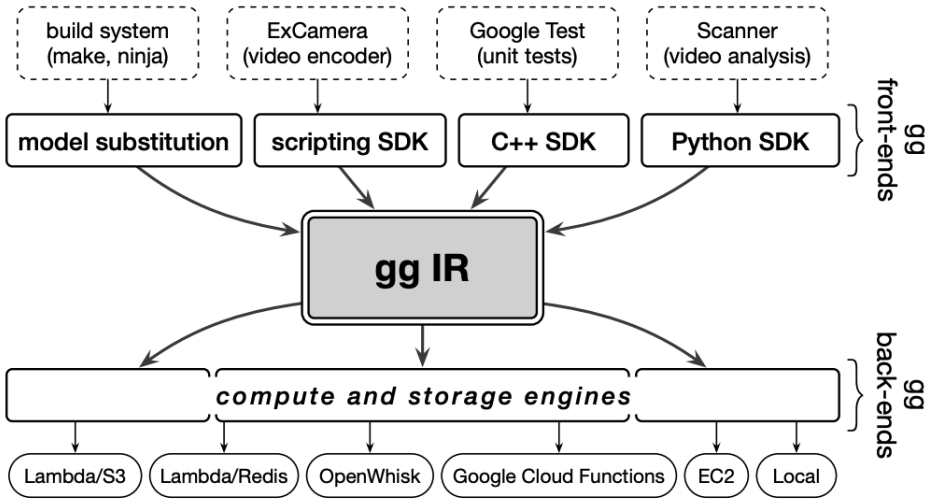

Figure 3.3: The *gg* engine [14].

Figure 3.3 is a high-level diagram that portrays how the *gg* engine works. The *gg* frontend engine builds the thunks, and the backend engine interacts with the cloud and makes sure the correct thunks are uploaded and downloaded.

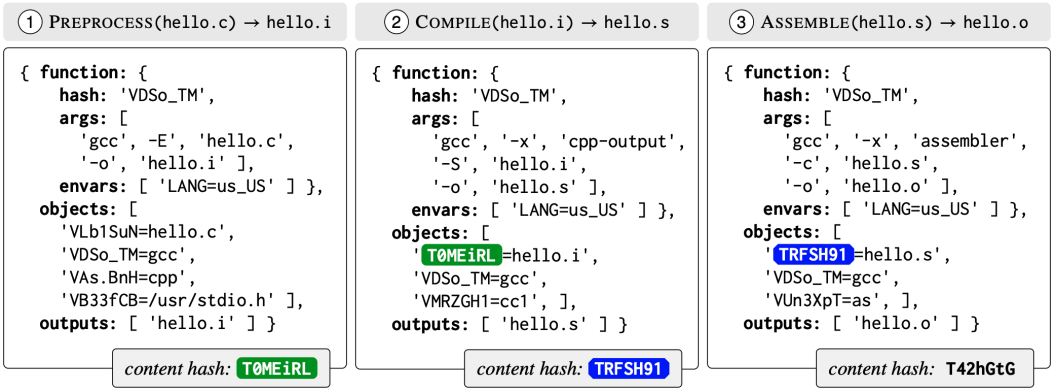

Figure 3.4: The *gg* build process [14].

Figure 3.4 is an example of how a Hello World is built using *gg*. *gg* needs all the above environment variables to generate a thunk. It creates a thunk for each process of the compilation process (preprocessing, compilation, assembling, and linking). Chapter 4 will discuss the steps required to build and run an application using *gg*.

3.7.2 Advantages Of Using *gg*

- *gg*'s engine works well with the cloud once it is set up.
- Allows for thousands-way parallelism
- Speeds up the build and execution of an application.
- Automatic scaling.

3.7.3 Disadvantages Of Using *gg*

- *gg* is still in active development and there's not a lot of documentation.
- Difficult to set up the environment when creating thunks.

3.8 Conclusion

In this Chapter different build systems and cloud technologies were discussed. The advantages and disadvantages of Docker and Kubernetes were also discussed.

Wake	**Bazel**	***gg***
Dependency linking and analysis		
Analyzes dependencies automatically as long as the input files are not under-specified.	Creates an action graph which is a list of all the dependencies and the steps needed to build a file.	Creates an intermediate representation of the target (thunk), and this is used to build the target.
Cloud Support		
No known support for building on the cloud yet.	Building projects on Google Cloud possible, but it is in the alpha stages.	Projects can be built on AWS and Google Cloud.
Code Extensibility		
Need to use the custom Wake language.	Need to use the custom Starlark language.	Extensibility can be done in C++ and Python.
OS Compatibility		
Compatible with Mac OS and Linux.	Compatible with Windows, Mac OS and Linux.	Compatible with Mac OS and Linux.

Table 3.3: Wake vs Bazel vs *gg*.

Table 3.3 compares the different build systems that are relevant to this project. Since LiveHD's codebase is written in C++, and *gg* is the only system that allows for building and running applications on the cloud, LiveHD's integration with *gg* makes the most sense, which will be discussed in the next chapter.

Chapter 4

Intergrating LiveHD With *gg*

> I knew exactly what to do. But in a much more real sense, I had no idea what to do.
>
> Michael Scott

This chapter will go over the best way to implement LiveHD functionality on the cloud. From the previous chapter, it is evident that using *gg* is the best way to perform passes on the cloud. There are a few requirements that need to be satisfied in order for the results to be as efficient as possible.

4.1 Requirements

- The main requirement is to make sure that all transformations can be scaled down to zero, which means that there shouldn't be any unnecessary processes running on the cloud. This would just take up resources, and in turn, cost money even if no work is being done.
- Making sure that passes can be done with parallelism, and that enables

multiple transformations to be done at the same time.

- The goal is not to build LiveHD using *gg*. That is already being handled by Bazel, but instead, the goal is to perform graph transformations on the cloud using *gg*.

4.2 Implementaion

gg works locally and on the cloud. On the cloud side, *gg* works with AWS Lambda as well as Google Cloud, but it was easier to set up AWS functions with *gg*, which is why the initial approach is to use *gg* with AWS.

4.2.1 Setup

The first step is to setup *gg* and figure out how it works. There are a few prerequisites to run *gg* on a local machine.

1. Cloning the *gg* repository `https://github.com/StanfordSNR/gg`
2. Making sure all the necessary libraries were installed such as gcc, g++, and python 3.
3. Setting up an AWS account and setting up an Amazon S3 storage bucket as well as the necessary AWS Lambda execution roles.
4. Setting environment variables on the local computer to make sure *gg* can communicate with AWS.
5. Installing the necessary *gg* functions on AWS.
6. Configuring *gg* with the necessary scripts to ensure it works.

4.2.2 Testing *gg*

These are three necessary commands to build and run an application with *gg*.

1. gg init - This creates a .gg directory so that the *gg* engine knows where the project files are.

2. gg infer - This command looks at the makefile in the project and generates a model (thunk) of all the dependencies needed to build and the run the application, as discussed in the previous chapter.

3. gg force - This command uses the *gg* engine and builds the model which was created by the previous step.

Initial testing was done on a few simple projects to make sure *gg* worked as expected.

Hello World

Starting off testing with a simple project is always a good idea. This project only consisted of 2 files, a source C++ file, and a makefile to build the file.

```
1 #include <iostream>
2 using namespace std;
3 int main(){
4     cout << "Hello World" << endl;
5     return 0;
6 }
```

Figure 4.1: A Hello World example

Figure 4.1 is a simple hello world example, and the makefile in Figure 4.2 is used to build the above application.

```
1 all:
2     g++ -o hello hello.cpp
3 clean:
4     rm hello
```

Figure 4.2: A Makefile for Hello World

When the application is built without *gg*, this is the result.

```
$ ./hello
Hello World
```

Figure 4.3: Running Hello World

Figure 4.3 shows the standard way to build a C++ application. To build the same Hello World application using *gg*, the following commands in Figure 4.4 need to be executed.

```
$ gg init

$ gg infer make
g++ -o hello hello.cpp
- generating model for g++
- input: hello.cpp
- generating make dependencies file... done.
- preprocessed: TNcYdYh09nO16LIq9bFIRmb3wgaCSZnfMf2Ee.eCPq9E000040f8
- compiled: TNJvHvCtncmxm7iaQGRD.WvU9IU4zrkWyG31DgktufTI0000028b
- assembled: TPrERTP4Y6O2nl.wQzaOaiGH4ap.uzxUtaQapFEgXpxY00000241
- linked: TLNvvv0HyY56ZFfwfzdSRQGl34tjDlC_vwrxqjDDKGPE00000ac0
- output: hello

$ gg force --engine=lambda --jobs=10 hello
- Loading the thunks... done (1ms).
- Uploading 1 file (98.0 B)... done (121 ms).
- Downloading output file (8.8 KiB)... done (93 ms).

$ ./hello
Hello World
```

Figure 4.4: Running Hello World using *gg*.

gg creates thunks for every step of the compilation (preprocessing, compilation, assembling, and linking. The thunk that gets generated in this case is also called hello, but it's not the final output. When the command "gg force" is used, it builds the thunk with *gg*'s engine. In the example above, the thunk was built on AWS Lambda with ten-way parallelism, however, there is an option to use *gg* locally. In order to do so, the "–engine=lambda" parameter can be changed to "–engine=local". Once the file has been built, the output file is downloaded and can be run by the user.

File I/O With C++

The next step of testing would be to make sure *gg* can execute a program which handles file input and output since that is one of the main goals of LiveHD, which is to perform graph transformations by reading input files and generating corresponding output files. Figure 4.5 is the current directory which contains all the files needed for this test.

```
$ ls
bin/ Makefile thunk_script.sh main main.cpp input output
```

Figure 4.5: Current files in the directory.

The goal here is not to build the project with *gg*, but instead try to execute the project with *gg* after it has been built. The program in Figure 4.6 handles file I/O with *gg*.

```
1  #include <string>
2  #include <fstream>
3  #include <iostream>
4  #include <cstdlib>
5  #include <libgen.h>
6  using namespace std;
7
8  int main(int argc, char* argv[]){
9          // Throw error if input file is not specified.
10         if (argc<2){
11                 cout << "Usage: ./main <input file>" << endl;
12                 return EXIT_FAILURE;
13         }
14         ifstream fin (argv[1]);
15         ofstream fout {"out"};
16         int input_number;
17         fin >> input_number;
18         if (!fin)
19                 fout << "error reading the file" << endl;
20         else
21                 fout << 2*input_number << endl;
22         return EXIT_SUCCESS;
```

Figure 4.6: A C++ program (main.cpp) that reads an input file and writes to an output file.

The program in Figure 4.6 takes character arguments with the second argument being the input file. The integer is read from the input file, multiplied by 2, and written to the output file. Since the executable, main, is not built using *gg*, there are a few other commands needed to execute the application using *gg*.

The first step is to generate a thunk, and the script from Figure 4.7 does what is required.

```
1  #!/bin/bash -e
2  USAGE="$0 <INPUT-FILE> <PROG-PATH>"
3  INPUT_FILE=${1?$USAGE}
4  PROG_PATH=${2?$USAGE}
5  PROG_HASH=$(gg-hash $PROG_PATH)
6  INPUT_HASH=$(gg-hash $INPUT_FILE)
7  gg init
8  gg-create-thunk --value $INPUT_HASH --executable $PROG_HASH --output
       out --placeholder out $PROG_HASH $PROG_HASH "@{GGHASH:$INPUT_HASH
       }"
9  gg-collect $PROG_FILE $INPUT_FILE
10 gg force --engine=lambda out
```

Figure 4.7: The script needed to generate a thunk for the program.

The script in Figure 4.7 asks for the input file from the user, creates thunks, and passes it to the C++ program. The tricky part is making sure that all the dependencies are included when creating this thunk. This includes hashing all the inputs as well as the executable. This way, the *gg* engine knows where to look for the necessary files.

After hashing all the files, "gg collect" can be used to collect those files. This ensures all the dependencies are accounted for. After all those commands are run, the command "gg force" is used to execute the application, and the final binary is built.

Integration With LiveHD

The next step is integrating *gg* with LiveHD. To make sure it works as expected, there needs to be a test case. This command would be run in the current, working LiveHD directory.

```
$ ./bazel-bin/main/lgshell "inou.yosys.tolg files:./inou/yosys/tests/
    trivial.v"
```

Figure 4.8: LiveHD Yosys InOu Test case.

The command from Figure 4.8 opens lgshell and passes the command to generate an LGraph from a Verilog file, which is trivial.v as discussed in Chapter 2. When that command is run, an lgdb directory gets generated and these are the graph files that exist in it.

```
graph_library.json  lg_1_type lg_data_node1nodename_k2vtxt
lg_data_npin1wirename_k2vtxt  lg_data_npin1wirename_v2ktxt lg_1_nodes
lg_data_node1nodename_k2v  lg_data_npin1wirename_k2v
lg_data_npin1wirename_v2k
```

Figure 4.9: Current list of files in lgdb.

Figure 4.9 shows the list of files in the lgdb directory after the command in Figure 4.8 is run. This was the expected result, and now this should be achieved with *gg*. The first step is to have it working locally. The first step is again, generating a thunk, and is done by script depicted in Figure 4.10. This script was inspired by one of the example projects in *gg*.

Local *gg* integration

It would be easier to run *gg* locally because LiveHD and all its dependencies would be installed on the local machine, and all the necessary file locations are known.

```
1  gg init
2  LG_PATH="./bazel-bin/main/lgshell"
3  MAIN_PATH="./main"
4  LG_HASH=$(gg-hash $LG_PATH)
5  MAIN_HASH=$(gg-hash $MAIN_PATH)
6  CUR_PATH="./inou/yosys/tests/trivial.v"
7  CUR_HASH=$(gg-hash $CUR_PATH)
8  COMMAND="inou.yosys.tolg files:.path/to/livehd/inou/yosys/tests/
       trivial.v"
9  gg-create-thunk --envar LG_FUNCTION_HASH=${LG_HASH} \
10                 --envar MAIN_FUNCTION_HASH=${MAIN_HASH} \
11                 --envar CUR_FUNCTION_HASH=${CUR_HASH} \
12                 --executable ${LG_HASH} \
13                 --executable ${MAIN_HASH} \
14                 --output lgdb \
15                 --placeholder lgoutput \
16                 ${MAIN_HASH} main ${COMMAND}
17 gg-collect $LG_PATH $CUR_PATH $TEST_PATH
18 gg force lgoutput
```

Figure 4.10: Bash script to generate a thunk.

The script shown in Figure 4.10 sets up the environment for the application in Figure 4.11 to be run using *gg*. All of these variables are needed for the *gg* engine to recognize what the executable is, and what the expected input and output files are. Even if there is one variable missing, the application will not run with *gg*.

```
1  #include <cstdlib>
2  #include <stdio.h>
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  using namespace std;
7
8  int main(int argc, char * argv[]){
9          if ( argc <= 1 ) {
10                 cerr << "Usage:./main <command being passed to
                       lgshell>"<< endl;
11                 return EXIT_FAILURE;
12         }
13         string input = "";
14         for (int i=1; i<argc; i++){
15                 input += argv[i];
16                 if(i!=argc-1)
17                         input += " ";
18         }
19         string finalCommand = "/home/rohan/Desktop/repo/livehd/bazel-
               bin/main/lgshell " + input;
20         system(finalCommand.c_str());
21         return EXIT_SUCCESS;
22 }
```

Figure 4.11: A C++ program written to open lgshell and pass a command.

The program in Figure 4.11 asks the user for an lgshell command, and then passes that command to lgshell. When the script from Figure 4.10 is run, it creates a thunk named lgoutput, and when this is run using *gg* force, it does create the lgdb directory locally, however, *gg* places the folder into a temporary directory

which it uses to execute the program. This means that it is possible to use *gg* with LiveHD locally. However running this same test case on the cloud is a much more difficult process, and the challenges faced will be discussed in Chapter 5.

4.3 Conclusion

The biggest challenge faced with integration was to account for all the possible dependencies that LiveHD. This was one of the main reasons why LiveHD doesn't currently work on the cloud, but it does work with the *gg* engine locally, which is a good proof of concept moving forward.

Chapter 5

Conclusion and Future Work

> I'm not superstitious, but I am a little stitious.
>
> Michael Scott

In this thesis, the possible solution to run LiveHD on the cloud was discussed. Chapter 2 discussed LiveHD and the need for fast simulation, and also explained the internals of LiveHD and the custom shell, lgshell.

Chapter 3 compared different cloud services offered today, and also went over how to containerize applications so that they could be deployed on the cloud. It also discussed different build systems that are relevant to this project.

Chapter 4 went over how to implement LiveHD on the cloud and the potential implementation in the future. Since *gg* is in active development, there are not a lot of resources available on how to use it. Once there is enough documentation on how to set up the right environment to generate thunks, integration should easier.

5.1 Things That Worked

- LiveHD integrated with *gg* locally.
- Better understanding of the cloud services and the *gg* engine, which is a promising tool in the world of cloud computing.

5.2 Improvements That Need To Be Made

- LiveHD did not work fully on the cloud because of several dependency issues. For example, if a Yosys command is being used, the Yosys graph library needs to be hashed by *gg*. All the input files and the necessary binaries should be hashed using the command "gg hash". This makes sure that the *gg* engine knows what files to use when running a command.
- Creating a git like system to apply patches, and making sure the input files can be transformed with parallelism, which means multiple files should be transformed at the same time instead of patching it in a serial order. This will be further discussed in the next section
- Creating static binaries for all the executables that are required when running a command. This is necessary because these binaries would not necessarily be running on the cloud, and there should be a way to generate a thunk that knows which binaries are needed for that command.
- A static binary for lgshell can be created by adding "linkopts = ['static']" in the Bazel BUILD file. However, creating a static binary for Yosys or Opentimer may not be that easy because they also depend on other dynamically linked libraries. The best way to link everything statically would be to indi-

vidually generate static binaries that lgshell depends on, and then generate a thunk using *gg*.

5.3 Need For Parallel Transformations

Currently, LiveHD supports serial transformations as seen in Figure 5.1. This means that files in the lgdb directory can only be transformed after the previous transformation has finished.

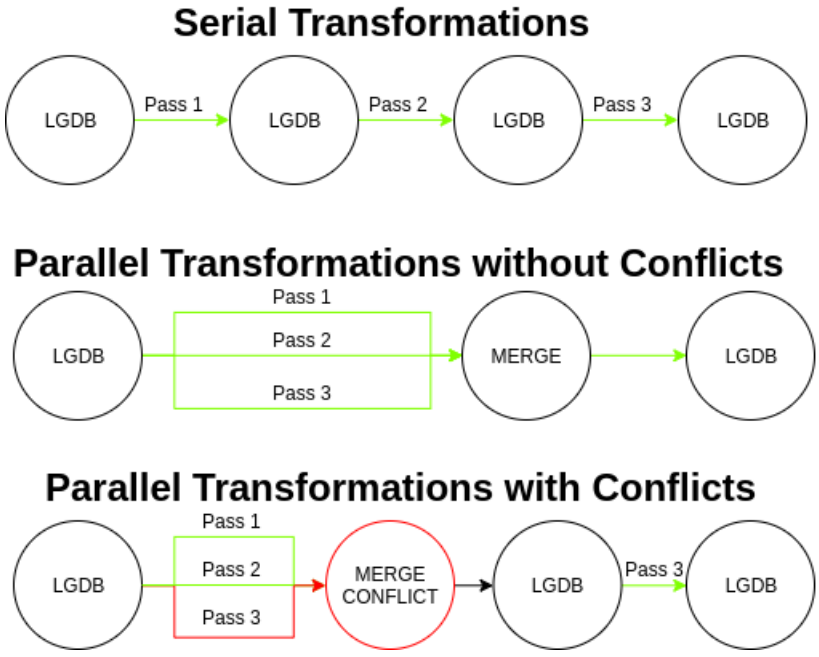

Figure 5.1: Serial vs Parallel LGraph transformations.

5.3.1 Goal

Serial transformations are not that efficient and there's no way to take advantage of any parallelism especially on the cloud. The goal is to find a way to perform multiple transformations to speed up the process.

Transformations are done by reading in the input files from the lgdb directory,

modifying them according to which command is passed, and then the files are saved. Serial transformations are safer because there won't be any conflict among transformations and the files will remain intact. However, to perform parallel transformations, there are a few challenges.

5.3.2 Challenges And Potential Solution

The main challenge to implement parallel transformations would be to make sure there aren't any conflicts when files are transformed. Figure 5.1 shows the flow needed to perform simultaneous transformations without conflicts.

The potential solution would be use a git-like system or a diff which can check which lines are being modified. For example in Figure 5.1, pass 2 and pass 3 try to modify the same line, which causes a conflict. This conflict can be resolved by letting pass 2 modify the file and then let pass 3 resubmit its synthesis results to that file using the updated input from pass 2.

This is similar to resolving merge conflicts when using git. As soon as there is a conflict, the command "git diff" can show the difference between the conflicting versions, and then the user can decide which version to use as a base and then update the code accordingly.

5.4 Final Remarks

I aim to keep working on the integration so that LiveHD can fully take advantage of cloud resources to produce even faster synthesis and simulation results. There is a better understanding of how *gg* works, and this creates a platform upon which further research can be done. All examples and scripts used in this project

can be found in the cloud branch of LiveHD under the *gg* folder.[1]

My main contribution to the LiveHD project was to find the most efficient way to run LiveHD on the cloud, and after a lot of groundwork, the best way to do so is to use *gg*. I tried using Docker and Kubernetes as well, but *gg* is more efficient because the engine makes it easier to create burst-parallel cloud functions from the source application, and these functions can be run on AWS or Google Cloud. This ensures that the application can take advantage of thousands-way parallelism, and instead of reinventing the wheel and trying to achieve the same functionality as *gg*, it is better to keep working on the integration of LiveHD and *gg*.

[1] *gg* - Examples and scripts. `https://github.com/masc-ucsc/livehd/tree/master/cloud/gg`

Bibliography

[1] LiveHD. https://github.com/masc-ucsc/livehd.

[2] Cloud adoption statistics for 2019. https://hostingtribunal.com/blog/cloud-adoption-statistics/.

[3] Rafael T. Possignolo, Sheng H. Wang, Haven Skinner, and Jose Renau. Lgraph: A multi-language open-source database. *Open-Source EDA Technology, Proceedings of the First Workshop*, Oct 2018.

[4] What is cloud computing? https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.

[5] What is docker? https://opensource.com/resources/what-docker.

[6] Docker. https://docker.com.

[7] What is kubernetes? An introduction to the wildly popular container orchestration platform. https://blog.newrelic.com/engineering/what-is-kubernetes/.

[8] Kubernetes - Production-Grade Container Orchestration. https://kubernetes.io/.

[9] What is kubernetes? https://www.redhat.com/en/topics/containers/what-is-kubernetes.

[10] Difference between kubernetes and docker. http://www.differencebetween.net/technology/difference-between-kubernetes-and-docker/.

[11] Aws vs google cloud platform. what is better for devops in the cloud? https://medium.com/@Iren.Korkishko/aws-vs-google-cloud-platform-what-is-better-for-devops-in-the-cloud-434a7cefa25a.

[12] Wake. https://github.com/sifive/wake.

[13] Bazel. https://bazel.build/.

[14] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. *2019 USENIX Annual Technical Conference*, Jul 2019.

[15] What is cloud computing? how does 'the cloud' work? `https://www.fastmetrics.com/blog/tech/what-is-cloud-computing/`.

[16] gg. `https://github.com/StanfordSNR/gg`.

[17] Rohan Prakash Granpati. OpenTimer Interface for LGraph . Master's thesis, University of California, Santa Cruz, June 2019.

[18] Kubernetes. `https://en.wikipedia.org/wiki/Kubernetes`.

[19] Aws vs google cloud: We asked the devs. `https://hackernoon.com/aws-vs-google-cloud-we-asked-the-devs-805q430bc`.